

LockPeeker: Detecting Latent Locks in Java APIs

Ziyi Lin¹, Hao Zhong^{2*}, Yuting Chen^{2*}, Jianjun Zhao³

¹School of Software, Shanghai Jiao Tong University, China

²Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

³Department of Advanced Information Technology, Kyushu University, Japan
{linziyi, zhonghao, chenyt}@sjtu.edu.cn, zhao@ait.kyushu-u.ac.jp

ABSTRACT

Detecting lock-related defects has long been a hot research topic in software engineering. Many efforts have been spent on detecting such deadlocks in concurrent software systems. However, *latent locks* may be hidden in application programming interface (API) methods whose source code may not be accessible to developers. Many APIs have latent locks. For example, our study has shown that J2SE alone can have 2,000+ latent locks. As latent locks are less known by developers, they can cause deadlocks that are hard to perceive or diagnose. Meanwhile, the state-of-the-art tools mostly handle API methods as black boxes, and cannot detect deadlocks that involve such latent locks. In this paper, we propose a novel black-box testing approach, called *LockPeeker*, that reveals latent locks in Java APIs. The essential idea of LockPeeker is that latent locks of a given API method can be revealed by testing the method and summarizing the locking effects during testing execution. We have evaluated LockPeeker on ten real-world Java projects. Our evaluation results show that (1) LockPeeker detects 74.9% of latent locks in API methods, and (2) it enables state-of-the-art tools to detect deadlocks that otherwise cannot be detected.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Latent lock; deadlock detection; API method

1. INTRODUCTION

In multi-threaded programs, locks are used to exclusively access memories. Defects such as deadlocks and livelocks can be raised when locks are inappropriately enforced and/or released by threads. As such defects decrease code quality, many approaches [4, 5, 13, 23, 28] have been proposed to detect locking-related defects. With the support of these approaches, many concurrency bugs have been detected, and some are previously unknown [13, 28].

*Corresponding co-authors

In modern programming, APIs are widely used, but their locks may not be well documented. Latent locks can live in API sources. In this paper, we refer to a lock located inside an API whose source is inaccessible as a **latent lock**. We have conducted a preliminary study by analyzing the source files of J2SE¹. The results show there are 2000+ locks behind J2SE APIs. These locks can be taken as latent locks if developers do not step into their sources.

Latent locks typically exist in two types of APIs: close-sourced APIs which are encrypted (*e.g.*, commercial libraries), and APIs whose programming languages are different from the programming languages of client code (*e.g.*, native C/C++/C#/Assembly APIs that are called through Java Native API). As developers may not be able to access the source code of encrypted or native APIs, it becomes impossible to find latent locks inside and detect defects related to these locks.

Motivating example and the state-of-the-art.

Latent locks can cause serious bugs such as deadlocks. Figure 1a shows a simplified version of Eclipse BIRT-287102² which contains a deadlock caused by latent locks. When a thread (`thread1`) of `Thread1` starts, it invokes the J2SE method `forName` (line 6); when a thread (`thread2`) of `Thread2` starts, it acquires a lock on `obj` (line 10). As shown in Figure 1b, a deadlock can happen, because the native method `forName0` acquires a lock on its parameter, `loader: thread1` can invoke `loadClass` and acquire a lock on `obj` (lines 14-15) when holding `loader`; at the same time, `thread2` invokes `forName` and acquires `loader` (line 11) when holding `obj`. However, even the thread dump (as Figure 1c shows) does not state that locks are held and acquired in the native method. Without any knowledge that a thread can lock a class-loader when invoking `forName0`, a program analysis tool cannot detect the deadlock.

To the best of our knowledge, previous deadlock detecting approaches are less effective in finding such bugs, since they omit analyzing APIs, and are often not comprehensive enough to analyze the cross language sources. For example, our previous work [20] shows that existing tools (*e.g.*, JPF [32] and CheckMate [13]) cannot detect deadlocks that involve latent locks in native code. As surveyed by Hong and Kim [10], to detect lock-related defects, all the existing approaches construct execution models for sources under analysis. For example, Sorrentino *et al.* [29] dynamically record reads and writes to shared variables, and construct an execution model for the program in analysis. However, when sources are unavailable, it is difficult to determine which variables are shared, and thus difficult to detect the corresponding concurrency bugs.

Some approaches [14, 26, 33] can analyze API methods, if their

¹<http://docs.oracle.com/javase/8/docs/api/index.html>

²https://bugs.eclipse.org/bugs/show_bug.cgi?id=287102

```

1. public class SimpleBirt287102 {
2.   private SimpleClassLoader loader;
3.   private Object obj;

4.   class Thread1 extends Thread {
5.     public void run() {
6.       Class.forName("java.lang.Object", true, loader);
7.     }
8.   }

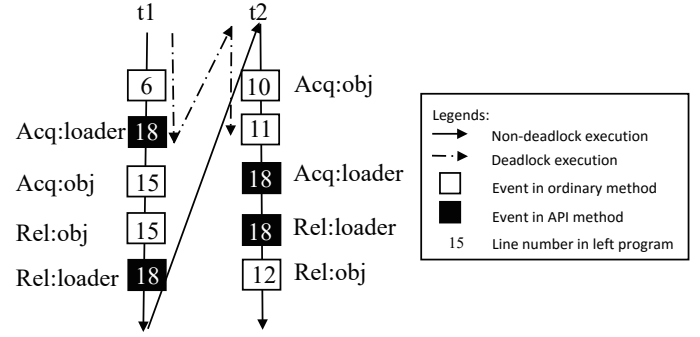
9.   class Thread2 extends Thread {
10.    public void run() {
11.      synchronized (obj) {
12.        Class.forName("java.lang.Object", true, loader);
13.      }
14.    }
15.  }

16.  class SimpleClassLoader extends ClassLoader {
17.    public Class<?> loadClass(String name) {
18.      synchronized (obj) { ... }
19.      return super.loadClass(name);
20.    }
21.  }

22.  private static native Class<?> forName0(String name,
23.    boolean initialize, ClassLoader loader, Class<?> caller);

```

(a) Buggy code



(b) Deadlock execution

```

"Thread-1":
  at java.lang.Class.forName0(Native Method)
  at java.lang.Class.forName(Class.java:348)
  at SimpleBirt287102$Thread2.run(SimpleBirt287102.java:45)
  - locked <0x00000000d5eee710> (a java.lang.Object)

"Thread-0":
  at SimpleBirt287102$SimpleClassLoader.loadClass(SimpleBirt287102.java:64)
  - waiting to lock <0x00000000d5eee710> (a java.lang.Object)
  at java.lang.Class.forName0(Native Method)
  at java.lang.Class.forName(Class.java:348)
  at SimpleBirt287102$Thread1.run(SimpleBirt287102.java:33)

```

(c) Thread dump

Figure 1: Buggy code simplified from BIRT-287102.

source code is available. Some other approaches generate replacements for API methods. For example, Shafiei and Breugel [27] manually write replacement code for native methods. As another example, Kalinovskiy [15] introduces a technique that decompiles Java bytecode. However, as a replacement is usually not precisely equivalent to the original method, a latent lock in the original API method may not appear in its replacement.

Our approach.

Having observed that API methods are not easy to analyze and believing that proper replacements can facilitate program analysis, we propose in this paper an approach named *LockPeeker* that leverages locking behavior’s effect in Java to construct a locking model for each API method. Such effect is reserved in any Java API, no matter whether it is encrypted or native. The model clearly presents whether some latent locks can be enforced in an API method, and thus aids program analysis tools in detecting deadlocks intertwined by explicit and latent locks.

However, there remain many difficulties of generating such locking behavior models, since *locked resources*, *locking structures*, and *locking conditions* are not explicitly defined, even in their API documents. Many related questions are raised: Are there locks in an API method? Which object(s) will be locked within the API method? What are the relations among locks? Are they nested? Is there any conditions for locking? Without answering these questions, it is not possible to construct a precise locking behavior model for an API method, since its details are still blind to analysis.

LockPeeker has two major steps to detect latent locks in Java API methods: (1) it repeatedly executes a target method and collects the locks triggered during execution, through which a lock tree structure is synthesized; and (2) locking conditions are inferred by observing which test inputs trigger the locks. Synthesized lock trees can be leveraged by existing tools for deadlock analysis.

This paper makes the following contributions:

- *Problem statement.* To the best of our knowledge, we are the

first to state the research problem of detecting latent locks in API methods. The research problem can motivate follow-up work that leads to more practical approaches and tools.

- *Approach.* We propose a novel approach, called LockPeeker, that detects latent locks in Java API methods. The essential idea of LockPeeker is that, given an API method, to perform an extensive unit testing of this method, through which (1) a tree of latent locks within the method can be derived by observing whether any locks can be enforced, and (2) the locking condition(s) can be inferred by learning which lock(s) can be acquired by which test input(s).
- *Implementation and evaluation.* We have implemented a tool to support LockPeeker and evaluated LockPeeker on ten real-world Java projects. Our results show that LockPeeker detects 74.9% of locks in API methods. Furthermore, LockPeeker allows locking trees and locking conditions to be generated for Java API methods, which helps find deadlocks that otherwise cannot be detected from such methods.

The rest is organized as follows. Section 2 introduce our preliminaries. Section 3 presents our approach. Section 4 evaluates LockPeeker. Section 5 surveys related work. Section 6 discusses relevant issues and the future work. Section 7 concludes this paper.

2. PRELIMINARIES

In this paper, we advocate the idea of inferring the locking structure and the associated conditions for a Java API method. In Java, a thread can use the `synchronized` keyword to acquire an object’s intrinsic lock that enforces exclusive access to the object, allowing a block of statements be exclusively executed. More sophisticated locking idioms (e.g., using a `Lock` object) are supported in the `java.util.concurrent.locks` package. For simplifying our discussion, we focus on intrinsic locks in this paper, because a `Lock` object can be interpreted as one or more intrinsic locks.

In a method, locks can be sequentially or nestedly acquired and

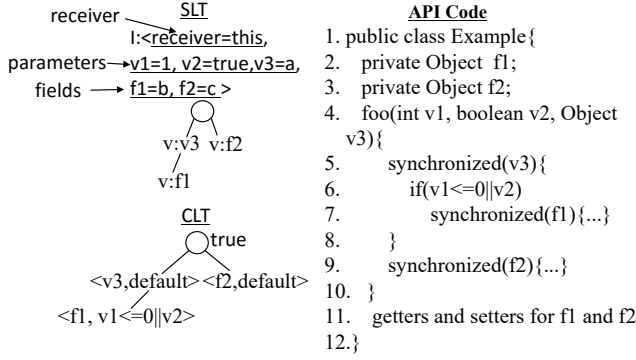


Figure 2: Example of SLT and CLT. The SLT represents the locking structure of the `foo` method. The CLT provides locking conditions. The getters and setters for the fields in API Code are omitted.

released, many times under certain conditions. Lock tree [8] and lock order graph [3, 21] have been proposed to describe how locks are organized in threads for deadlock detection. A lock tree is typically related to a thread, and its nodes are stateless: its root represents the thread, and each node represents a lock acquired by the thread. We use a similar tree to represent locking structure and locking conditions in a method. Moreover, we need stateful information (*i.e.*, test inputs of the target method) on the tree to infer the locking conditions.

We use a *test input interface* to collect all variables used for testing a target method. Correspondingly, a *test instance* is composed of the concrete values for a *test input interface* in a test session. We extend a lock tree to a stateful lock tree (SLT).

DEFINITION 1 (STATEFUL LOCK TREE). *For a method (m) declared in a class (C), a stateful lock tree describes, when a lock is triggered, the execution structure and the test instance of m .*

- The root (I) denotes a test instance that triggers the lock, including the receiver (the `this` object or $C.class$), the parameters of m and the fields of C .
- A node $\langle v \rangle$ denotes the variable that contributes the lock.
- A parent-child relation between two nodes indicates that two locks are nestedly enforced: the parent corresponds to the outer lock, and the child the inner one.

The locks can be conditionally enforced. We thus use condition lock tree (CLT) to represent such a locking structure and the locking conditions in a method:

DEFINITION 2 (CONDITION LOCK TREE). *For a method, its CLT is an SLT whose nodes are supplemented with conditions.*

- The root (con) denotes the default condition for all nodes.
- A node $\langle v, con \rangle$ denotes the variable on which the lock is placed and the condition to trigger the lock.

A child can be associated with a condition stronger than its parent's condition. The condition on root is by default `true`, indicating that the lock is triggered unconditionally. Any condition on a node is by default the same as the condition on its parent.

For example, Figure 2 shows an SLT and a corresponding CLT derived from the `foo` method. The root of the SLT contains a test instance consists of receiver, method parameters and class fields. Node $v3$ is `foo`'s third parameter and Nodes $f1$ and $f2$ are class fields, where $v3$ and $f1$ are nested locks represented by the left

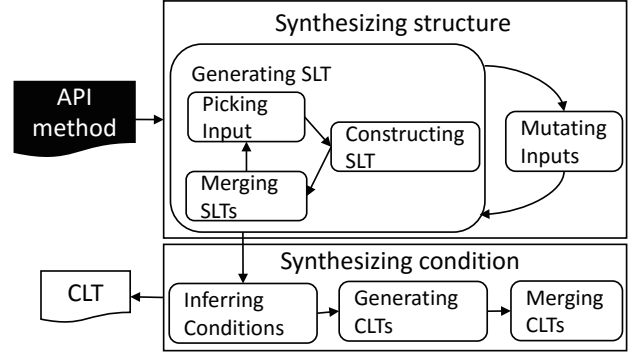


Figure 3: Overview of LockPeeker.

subtree, and $f2$ is a lock separately represented by the right subtree. The locking condition is shown on the CLT. Node $f1$ has a specified condition, indicating the variable $f1$ is locked under the condition $(v1 \leq 0 || v2)$, and other nodes' conditions are by default `true`, indicating that they are locked unconditionally. We explain more details of SLTs and CLTs in Section 3.2.

3. APPROACH

This section first presents an overview of LockPeeker (Section 3.1), and then presents how to synthesize the structure of a CLT (Section 3.2) and how to infer the conditions of the CLT (Section 3.3).

3.1 Overview

LockPeeker constructs a CLT describing the locks for a target method m . As shown in Figure 3, LockPeeker takes two major steps in constructing a CLT:

Step 1: Constructing SLTs. LockPeeker iteratively generates test instances and executes m . At each iteration, LockPeeker checks whether locks are triggered in m : If a test instance can trigger locks, an SLT is constructed to represent the observed locks. In this step, we can generate many SLTs for a given API method, each corresponding to one test instance.

Step 2: Merging SLTs into a CLT. LockPeeker merges the SLTs into a CLT whose nodes' conditions are inferred by learning the test instances. The CLT represents the observable locks and locking conditions in m , allowing a program replacement to be created. The program replacement allows existing tools to step into m during locking analysis.

3.2 Synthesizing Structure

3.2.1 Constructing SLTs

For a target API method m , LockPeeker constructs an SLT for m through performing a unit testing of the method. We formalize a detection process as follow:

$$slt = par(thread_1.acquire(i), thread_2.call(m, I)) \quad (1)$$

where *par* requires that the two threads shall run in parallel; I denotes a test instance; i denotes a variable of I ; and *slt* denotes a SLT that describes the observed locks in the execution. A simple construction involves two threads: $thread_1$ is a thread acquiring a lock on i , $thread_2$ another calling m . Once $thread_2$ is blocked to wait for locking i , i is the object locked in m . Therefore, LockPeeker firstly starts $thread_1$ and then $thread_2$, and observes whether $thread_2$ is blocked while invoking m , *i.e.*, whether i is locked within m .

Algorithm 1 Constructing an SLT

Input: A test instance I and an API method m **Output:** An SLT with locks ret

```
1:  $ret \leftarrow root(I)$ 
2: for each  $i$  in  $I$  do
3:    $thread_1.start$ 
4:    $lock(i)$ 
5:    $thread_1.suspend$ 
6:    $thread_2.start$ 
7:    $call\ m(I)$ 
8:   if  $thread_2$  is blocked on acquiring  $i$  then
9:      $tree \leftarrow tree(I)$ 
10:     $nestingLocks \leftarrow checkNesting(t_2)$ 
11:    if  $nestingLocks \neq \emptyset$  then
12:       $tree.add(nestingLocks)$ 
13:    else
14:       $tree.add(i)$ 
15:    end if
16:     $ret \leftarrow ret.merge(tree)$ 
17:  end if
18:   $thread_1.terminate$ 
19:   $thread_2.terminate$ 
20: end for
```

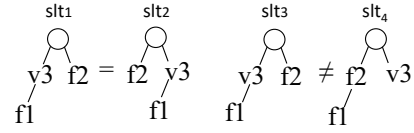
Algorithm 2 $cur.merge(target)$

Input: Current SLT, cur Objective SLT to merge, $target$ **Output:** Current SLT merged with the objective SLT

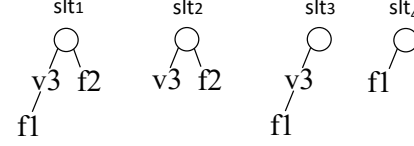
```
1: if  $isSimple()$  then
2:   return corresponding SLT
3: end if
4: for each  $targetChild$  in  $target.children$  do
5:    $node2Add \leftarrow null$ 
6:   for each  $curChild$  in  $cur.children$  do
7:     if  $targetChild.isSame(curChild)$  then
8:        $node2Add \leftarrow null$ 
9:        $curChild \leftarrow curChild.merge(targetChild)$ 
10:      break
11:    else
12:       $node2Add \leftarrow targetChild$ 
13:    end if
14:  end for
15:  if  $node2Add \neq null$  then
16:     $cur.add(node2Add)$ 
17:  end if
18: end for
19: return  $cur$ 
```

When two non-nested locks are observed, we cannot determine which lock is ahead of another, as we do not have the source of m . However, for the purpose of deadlock detection, only the nested locking sequence matters.

Algorithm 1 shows an iterative process of constructing for a method an SLT *w.r.t.* a test instance. It creates the SLT's root (ret) based on the test instance I (line 1). For each variable i in I , $thread_1$ is started to acquire a lock on i . After the lock is acquired, $thread_1$ suspends and keeps holding it (lines 3-5). $thread_2$ is then started to invoke m with I as its input values by reflection (lines 6-7). The two threads are observed (by the *main* thread). Line 8 checks whether $thread_2$ is blocked, and more importantly, whether $thread_2$ happens to be blocked by the intrinsic lock that is being held by



(a) SLT equivalent relation. slt_1 and slt_2 are equivalent, but slt_3 and slt_4 are not.



(b) SLT containing relation. slt_1 contains slt_2 and slt_3 , but slt_2 is not a subtree of slt_1 , while slt_3 is. slt_4 is not contained in slt_1 .

Figure 4: Equivalent and containing relations of SLTs. Only structures are considered in these two relations, values of test instances in the roots are ignored.

$thread_1$, as $thread_2$ may also be blocked for the other reasons (e.g., an invocation of the `wait()` method in m). If so, an empty SLT $tree$ is created (line 9) and `checkNesting`³ is called to check whether there exist nested locks (line 10). If there are nested locks, they are all added into $tree$ (lines 11-12). Only the current checking object i is added otherwise (line 14).

After that, $tree$ is merged into ret (line 16). The merging of SLTs can be performed by checking whether an SLT can be equivalent to another or contained by another:

RELATION 1 (SLT EQUIVALENT). *Two SLTs are structurally equivalent, regardless of the test instance, denoted as $slt_1 \doteq slt_2$, as shown in Figure 4a.*

RELATION 2 (SLT CONTAINING). *One SLT slt_1 contains another slt_2 , denoted as $slt_2 \subseteq slt_1$, if there exists an slt_1 's subtree which can be added to a certain node of slt_2 's such that $slt_1 \doteq slt_2$, as shown in Figure 4b.*

Algorithm 2 shows how function `merge()` works. It firstly checks if the two candidate SLTs to merge satisfy Relation 1 or Relation 2, and returns cur for Relation 1 or the containing SLT for Relation 2 (lines 1-2). Otherwise, it moves on to traverse $target$'s children nodes and compares each one (say $targetChild$) with each of cur 's child node (say $curChild$). If $targetChild$ and $curChild$ are same (nodes $node_1\{v_1\}$ and $node_2\{v_2\}$ are same when $v_1 = v_2$), their subtrees are merged recursively (lines 7-10). If no same node can be found, $targetChild$ is added as a new child (lines 11-18).

We next use the `foo` method in Figure 2 to illustrate the merge function. Let two SLTs ($tree1$ and $tree2$) be merged: $tree1$ has only one node ($v3$), and $tree2$ has two parent-child nodes ($v3$ and $f1$). As $tree1 \subseteq tree2$, the merge function returns $tree2$ and eliminates a redundant tree (i.e., $tree1$). If $tree1$ has only one node ($f2$), the merging of $tree1$ and $tree2$ results the SLT in Figure 2.

Picking up detection candidates. A method may have a large set of variables in its test interface as locking candidates. For a method m declared in a class C , the locking candidates can be categorized into four sets: S_{mp} (the parameters of m), S_i (the receiver, the `this` object, and the class object of C), S_f (the fields of C) and

³Function `checkNesting()` takes the blocked thread as input. It retrieves the thread's information by calling `getThreadInfo()` method from `ThreadMXBean`, an interface available since JDK 1.6.

S_e (the environment objects). Our empirical study which will be explained in Section 4.4 clearly shows that locks on environment objects and parameters are rare, while it will be quite difficult to explore all environment variables *w.r.t.* black box APIs. Thus we omit S_e , but leave it for future work. In our study we consider only the following sets as locking candidates:

$$S_c = S_{mp} \cup S_i \cup S_f. \quad (2)$$

LockPeeker also omits variables whose types are primitive (*e.g.*, Integer and Float), since it is error-prone to lock such resources. Indeed, many programmers believe that it is a bad practice to acquire locks on variables of primitive types. For example, Findbugs [12] determines such locks as bugs⁴. Therefore, we assume developers follow the good practise, but spend more resources on checking the locks on non-primitive variables for efficiency.

3.2.2 Mutating test instances

Algorithm 1 constructs an SLT from an initial test instance which is related to a set of default values for its variables: *e.g.*, 0 for int variables, and true for boolean variables. For complex types, they are instantiated with their simplest constructors with the support of reflection. If such variables do not have visible constructors, mocked objects are then taken, with the support of Mockito⁵.

Meanwhile, using the default values can detect the structures of simple locks (*e.g.*, the `forName0` method), but is insufficient for those methods with complicated structures (*e.g.*, the `foo` method in Figure 2). As test instances with different values for their variables can reveal locks in different branches, we mutate the initial test instance to explore complicated structures.

However, it is infeasible to completely determine how many mutants are sufficient: too few mutants may not explore all necessary branches; meanwhile, it is expensive to use a number of mutants to test the method, if few conditional locks exists within the method. LockPeeker takes a two-phase exploration strategy to tackle this problem.

LockPeeker first takes a *blind exploration* phase to detect (1) whether the locks are conditional, and (2) what their structure(s) are. The variable values of a test instance are randomly mutated for exploring branches. LockPeeker reuses the mutation operators that are proposed by Alexander *et al.* [1]. Besides mutating individual values, LockPeeker considers a combination of values. For example, if an value is of a *boolean* type, LockPeeker keeps it as true or false, and tries values for other variables. The blind exploration needs to terminate in a limited number of iterations. We have evaluated the blind exploration phase (the details will be explained in Section 4.3), and the results show 100 is a reasonable threshold to terminate the exploration.

Next, LockPeeker takes a *guided exploration* phase in which a greedy heuristic is used to search for more SLTs for condition boundaries. As the locking condition's structure has been detected, in this phase only the variables that are involved in the condition are mutated. The rationale of this is based on an assumption that SLTs are neighbors. Thus it is feasible to find more SLTs by exploring the neighbors of a found SLT. For example, if the root of SLT_m is an integer whose value is m , LockPeeker first tries $m - n$ and $m + n$, where n is randomly picked. If two STLs (SLT_{m-n} and SLT_{m+n}) are built, LockPeeker compares the built STLs for the next mutation: if $SLT_{m-n} \doteq SLT_m \neq SLT_{m+n}$, LockPeeker takes a fast mutation strategy to look for the boundary which lies between m and $m + n$; if $SLT_{m-n} \neq SLT_m \doteq SLT_{m+n}$, the

boundary value is explored between $m - n$ and m . It explores the boundary value in both directions by enlarging n otherwise.

Mutation strategy. LockPeeker takes a fast mutation strategy for exploring the boundary values effectively.

Let two SLTs (S_1 and S_2) be inequivalent, and their test instances be $I_1 = \{i_1^1, i_2^1, \dots, i_n^1\}$ and $I_2 = \{i_1^2, i_2^2, \dots, i_n^2\}$. We search for the boundary value, $I' = \{i_1', i_2', \dots, i_n'\}$, between I_1 and I_2 . During the search, LockPeeker changes the numeric values to their averages, but does not change the nominal values. Based on the new test instance I_3 , LockPeeker builds a new SLT (S_3). If S_3 equals to S_1 , LockPeeker searches between I_3 and I_2 ; if S_3 equals to S_2 , LockPeeker searches between I_3 and I_1 . The search process continues until ΔI is close to zero.

3.3 Synthesizing Condition

After exploring boundary values by mutating inputs, LockPeeker generates many discrete SLTs. It infers conditions and deduces a CLT out of those generated SLTs to represent the locks of an API method. For simplicity, we assume that an input is compared with only an operator (*e.g.*, $>$, $<$, and $==$), and the values to compare are constants.

Inferring a decision tree LockPeeker uses a decision tree [25] to classify SLTs. In a decision tree, a leaf represents a tree structure that is shared by a number of SLTs; the path from the root to a leaf represents a classification rule for the tree structure. LockPeeker infers the decision tree with the C4.5 algorithm implemented in an open source machine learning tool Weka [7]. The algorithm takes encoded SLTs as its inputs, and infers a decision tree. An SLT is encoded by its tree structure and inputs, where the tree structure is used for classifying and the input values for inferring rules for each classification.

Constructing CLTs Based on the decision tree, LockPeeker constructs CLTs. A tree structure can be restored from a leaf of a decision tree (*DT*), and the condition for such tree structure be deduced from the path from *DT*'s root to the leaf. The CLT is constructed by putting the condition to the root of the tree structure. Multiple leaves can lead to the same tree structure, indicating the condition for such a CLT is a union of all the paths.

Merging CLTs Algorithm 3 describes how CLTs are merged. Most of the algorithm is similar to Algorithm 2 except that: (1) no checks for Relation 1 or Relation 2 are performed at the beginning, because nodes conditions are still required to merge even if any of the relations is satisfied; (2) conditions are combined with an `or` operator (line 6) when two nodes are same (nodes $node_1(v_1, con_1)$ and $node_2(v_2, con_2)$ are same, if $v_1 = v_2$).

Figure 5 shows an example of the whole process of revealing latent locks in the `foo` method in Figure 2. It starts from an initial test instance which is then iteratively mutated to generate SLTs. Two SLTs ($tree_1$ and $tree_4$) have two nodes, and the other two SLTs ($tree_2$ and $tree_3$) have three nodes, more SLTs with similar structures are not shown. After they are encoded, $tree_1$ and $tree_4$ are grouped into $locktree_1$, and $tree_2$ and $tree_3$ are grouped into $locktree_2$. LockPeeker infers a decision tree with three leaves from encoded SLTs. The right two leaves refer to $locktree_2$, and the left leaf refers to $locktree_1$. Based on the decision tree, LockPeeker constructs two CLTs with conditions specified on roots. $locktree_1$'s condition is parsed from the decision tree's leftmost path, $locktree_2$'s condition is parsed from the other two paths. At last, two CLTs are merged into one and merged conditions are simplified (*i.e.*, $f1$'s condition is simplified to $v1 \leq 0 \vee v2 == \text{TRUE}$, and the other nodes' conditions true).

⁴<http://findbugs.sourceforge.net/bugDescriptions.html>

⁵Mockito is a popular mocking framework for unit testing of Java programs, see <http://site.mockito.org/>

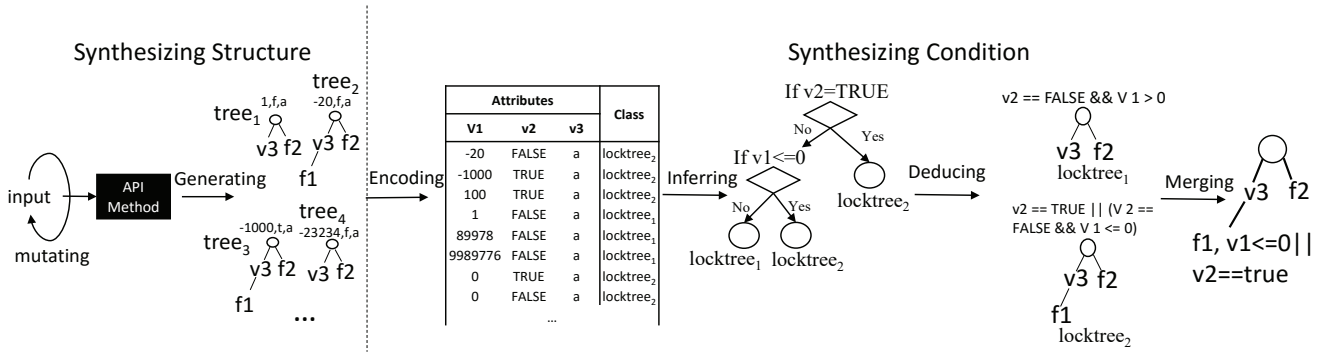


Figure 5: An example of the whole process to construct a CLT from a target API method. Only method parameters are shown for simplicity.

Algorithm 3 cur.cMerge(target)

Input: Current CLT, *cur*

Objective CLT to merge, *target*

Output: Current CLT merged with the objective CLT

```

1: for each targetChild in target.children do
2:   node2Add ← null
3:   for each curChild in cur.children do
4:     if targetChild.isSame(curChild) then
5:       node2Add ← null
6:       curChild.condition ← curChild.condition ∨ targetChild.condition
7:       curChild ← curChild.cMerge(targetChild)
8:       break
9:     else
10:      node2Add ← targetChild
11:    end if
12:  end for
13:  if node2Add ≠ null then
14:    cur.add(node2Add)
15:  end if
16: end for
17: return cur

```

4. EVALUATION

We have developed a tool for LockPeeker, and conducted evaluations on ten open source projects. The evaluation is to answer the following four research questions:

- **RQ1:** How effective is LockPeeker in revealing locks in Java API methods (Section 4.1)?
- **RQ2:** What kinds of deadlocks can be detected, if our detected latent locks are integrated (Section 4.2)?
- **RQ3:** What is the significance of LockPeeker’s threshold (Section 4.3)?
- **RQ4:** What are the essential test instance variables that may trigger locks (Section 4.4)?

Correspondingly, our evaluation results, which will be explained in this section, indicate that (1) LockPeeker detects 74.9% locks from real-world project methods; (2) with our detected latent locks, existing tools are able to detect real deadlocks that involve latent locks in native methods; (3) when the threshold is 100, we have 90.8% of confidence that the conditions and structures of locks can be fully revealed; and (4) a majority of locks are placed on method parameters, fields, and receivers.

4.1 RQ1. Detected Locks

Table 1: Subjects. In this table, M and EM represent the total number of methods and the number of evaluated methods, respectively; L and EL represent the total number of locks and the number of evaluated locks, respectively.

Project	LOC	M	EM	L	EL	Version
DBCP	5,792	6	6	6	6	1.2
Derby	357,575	535	34	581	37	10.5.1.1
FtpServer	12,039	7	7	8	8	1.0.6
Groovy	119,586	42	22	44	23	1.7.9
Hsqldb	165,787	97	30	105	32	2.3.3
Log4j	15,615	39	13	43	15	1.2.15
Lucene	45,842	104	21	126	24	2.9.3
Pool	1,891	13	10	13	10	1.2
Tomcat	218,882	417	33	464	35	8.0.29
Xalan	12,039	23	13	24	13	2.7.2
Total	955,084	1,283	189	1,414	203	

4.1.1 Setup

Subjects Ten popular open source projects used in concurrency testing researches [20] are employed as our subjects (Table 1). LockPeeker takes their methods as API methods, and does not analyze their code. The ground truth is established by a code reading of these projects. We firstly searched in the ten projects the synchronized keywords that are enforced within methods, and in total, 1,414 synchronized keywords were found in 1,283 methods. The results were analyzed by three software engineering graduate students to identify the sources of locking objects, the locking structures and locking conditions. This work is manually completed, as few tools can just meet our requirements.

As discussed in Section 3.2.1, we focus on locking candidates from method parameters, fields and receivers. To guarantee the representativeness, we selected locks from all of the three types, which include 189 methods with 203 locks: All of the 47 methods that acquire 49 locks on method parameters were selected, and for fields and receivers, we randomly selected ten methods from each project. We thus obtained 77 methods that acquire 86 locks on fields, and 67 methods that acquire 68 locks on receivers. There are two overlapping methods for parameters and fields.

Table 1 shows our subjects. Columns “Project”, “Version”, “LOC”, “M”, “EM” list their project names, versions, project sizes, the numbers of methods that have locks, and the number of methods used in the evaluation respectively. Columns “L” and “EL” list the numbers of synchronized keywords inside methods and those of evaluated locks, respectively.

Metrics We compared our detected locks with the ground truth at three levels: (1) whether a lock is detected; (2) whether the relations (sibling, nesting) among locks are detected, where the sibling sequence is ignored; and (3) whether the branching specifications

Table 2: Results of detecting locks.

Projects	Parameter			Field			Receiver			Total		
	RC_1	RC_2	#Locks	RC_1	RC_2	#Locks	RC_1	RC_2	#Locks	RC_1	RC_2	#Locks
DBCP	N/A	N/A	0	25%	25%	4	100%	100%	2	50%	50%	6
Derby	78.6%	78.6%	14	84.6%	84.6%	13	80%	80%	10	81.1%	81.1%	37
FtpServer	25%	25%	4	50%	50%	4	N/A	N/A	0	37.5%	37.5%	8
Groovy	0%	0%	3	90%	90%	10	70%	70%	10	69.6%	69.6%	23
HsqlDB	75%	75%	12	80%	100%	10	80%	100%	10	78.1%	90.6%	32
Log4j	0%	0%	1	83.3%	83.3%	12	0%	100%	2	66.7%	80%	15
Lucene	100%	100%	1	75%	83.3%	12	18.2%	54.5%	11	50%	70.8%	24
Pool	N/A	N/A	0	N/A	N/A	0	80%	90%	10	80%	90%	10
Tomcat	71.4%	71.4%	14	63.6%	63.6%	11	80%	90%	10	71.4%	74.3%	35
Xalan	N/A	N/A	0	40%	40%	10	100%	100%	3	53.8%	53.8%	13
Total	65.3%	65.3%	49	70.9%	74.4%	86	67.6%	82.4%	68	68.5%	74.9%	203

(e.g., condition expressions, catch clause, and loops) are detected. When all tree levels are reached, we say that the lock is *strictly detected*. When the first two levels are reached, the lock is *loosely detected*. Meanwhile, even if a detected lock is loosely correct, it helps reveal deadlocks.

The definitions of our recall are:

$$RC_1 = \frac{\text{strictly_detected}}{\text{total}} \quad (3)$$

$$RC_2 = \frac{\text{loosely_detected}}{\text{total}} \quad (4)$$

where *strictly_detected* (or *loosely_detected*) denotes the number of the strictly (or loosely) detected locks, and *total* denotes the total number of locks.

Default call sequence and input values To reveal locks, a method shall be invoked with appropriate call sequences and values. Typically, tools analyze sources to obtain call sequences and values. For example, GRT [22] can generate correct call sequences. But such techniques are not ready for API methods whose source are inaccessible. We thus use the default values for primitive objects, instantiate complicated objects by their default constructors or mocked objects, but omit call sequences.

4.1.2 Result

LockPeeker does not report any false positives in the evaluation. We manually compare the reported locks with the ground truth. All reported locks reside in the subject programs.

Table 2 shows the recalls of our detected locks. Columns “Parameter”, “Field”, and “Receiver” list detected locks on method parameters, fields, and receivers, respectively. Column “Total” lists total detected locks. Sub-columns “ RC_1 ”, “ RC_2 ”, and “#Locks” list the RC_1 s, RC_2 s, and the number of locks, respectively.

The results show that (1) LockPeeker detects 74.9% of locks in total, and (2) it effectively detects all of the three types of locks. We inspected the inferred locks, and found that some inferred locks are complicated. For example, the `forceFlush` method in Derby has a nested lock when a boolean field is `false`. Below shows the relevant code:

```

if (stopShipping) return;
synchronized (forceFlushSemaphore) {
    synchronized (objLSTSync) {...}
    ...
}

```

LockPeeker successfully detects structures and conditions of many such locks. However, it fails in detecting 49 locks due to two main reasons, the details shall be discussed in Section 6.

- *Insufficient call sequences and input values.* 42 out of 49 locks are not detected, since special call sequences and input

values are needed to trigger these locks. As API methods are in black boxes, it is infeasible to explore call sequences and input values.

- *Complicated code structures.* 7 out of 49 locks are not detected, since their structures and conditions are rather complicated.

In addition, as discussed in section 3.2.1, LockPeeker omits searching for the locks on variables of primitive types. It is worth noticing that programmers can enforce locks on such variables in practice, which violates our initial assumption. For example, the programmers of the Xalan project acquire locks on variables of the Boolean types.

In summary, our results lead to the first observation: LockPeeker is able to detect more than two thirds of locks in real-world APIs, even if their sources are inaccessible. In addition, the variety of the subjects in Table 2 is high. Although LockPeeker is less effective in detecting locks from complicated methods, these complicated methods are not evenly distributed in the selected subjects.

4.2 RQ2. Detected Deadlocks

4.2.1 Setup

We integrate LockPeeker with a deadlock detection tool named CheckMate whose idea is originally proposed by Joshi *et al.* [13] and reimplemented in our previous work [20]. CheckMate instruments source code of a program under analysis to collect its concurrency behaviors (including *synchronizing*, *starting*, *joining*, *waiting*, and *notifying* threads). After executing the instrumented program, CheckMate records a trace program, which is an execution model of the original program. CheckMate then employs JPF to detect deadlocks from the execution model. We compared the capabilities of CheckMate in detecting deadlocks, before and after it is integrated with LockPeeker.

Subjects. Table 3 shows the descriptions and the repair time of our selected deadlocks. The following call sequences trigger deadlocks with latent locks:

1. The New Relic bug of Jboss.⁶ The native method `forName0` has a latent lock.

- Thread1: `lock(forName0's parameter, ModuleClassLoader) → lock(Verifier);`
- Thread2: `lock(Verifier) → lock(ModuleClassLoader).`

2. IBM IV-30066.⁷ The native method `forNameImpl` has a latent

⁶<https://discuss.newrelic.com/t/jboss-7-1-1-crashes-with-deadlock/408>

⁷<http://www-01.ibm.com/support/docview.wss?uid=swg1IV30066>

Table 3: Found real-world deadlock bugs that involve latent locks.

Bug	Bug Description	Repair Time
New Relic with Jboss	JBoss does not start up correctly because of a deadlock in 3 threads when New Relic is enabled.	Unknown
IBM IV-30066	In JVM class loading, there is a validation whether a class is already in loading process or not. But it may deadlock during this validation.	7 days
Eclipse BIRT-287102	A deadlock happens when a data import thread opens a lot of files in a loop and another thread attempts to initialize a BIRT report engine in the same time.	7 days
Groovy-4736	When multiple threads uses <code>GroovyClassLoader</code> to get Groovy classes or just work with them and an other(s) threads change the sources, then deadlock can happen.	3 years

```

Found locks in the method
java.lang.Class.forName0(Ljava/lang/String;
    ZLjava/lang/ClassLoader;Ljava/lang/Class;)

synchronized (p3) {
}
p3: the third method parameter

```

Figure 6: An example of latent lock reported by LockPeeker.

lock.

- Thread1: `lock(forNameImpl's parameter, ExtClassLoader) → waiting Thread2 terminates;`
- Thread2: `lock(ExtClassLoader) → terminate).`

3. Eclipse BIRT-287102.⁸ The native method `forName0` has a latent lock.

- Thread1: `lock(forName0's parameter, ChildFirstURLClassLoader) → lock(JarFile);`
- Thread2: `lock(JarFile) → lock(ChildFirstURLClassLoader).`

4. Groovy-4736.⁹ The two native methods `forName0` and `getDeclaredFields0` both have latent locks.

- Thread1: `lock(HashMap) → lock(GroovyClassLoader$InnerLoader in getDeclaredFields0);`
- Thread2: `lock(forName0's parameter GroovyClassLoader$InnerLoader) → lock(HashMap).`

4.2.2 Result

CheckMate originally detects none of the four deadlocks. After it is integrated with LockPeeker, CheckMate detects all the four deadlocks, since LockPeeker supplements CheckMate with the latent locks inside the unanalyzable API methods.

Example We use BIRT-287102 in Figure 1 to illustrate how our detected locks improve CheckMate.

CheckMate alone does not report the deadlock in BIRT-287102, as it is not aware of the existence of latent locks inside the native method `forName0`. The trace program (Figure 7a) generated from the buggy code (Figure 1a) shows that the two threads both request the lock on `obj1`. CheckMate alone does not report any deadlock for this because there is actually no cyclic locking behaviors in the trace program.

LockPeeker supports CheckMate with support in finding potential deadlock in two steps:

- Detecting latent locks. LockPeeker scans all the API methods that are called in the buggy code, and finds that the `forName0` method has a latent lock on its third method parameter which is named as `p3`. For the native method, LockPeeker generated a CLT that has only one node, `p3`, indicating that the method has a latent lock on its third parameter with condition `true`. Figure 6 shows the reported latent locks.

⁸https://bugs.eclipse.org/bugs/show_bug.cgi?id=287102

⁹<http://jira.codehaus.org/browse/GROOVY-4736>

```

public class TraceProgram {
    static Object obj1 = new Object();

    static Thread t1 = new Thread() {
        public void run() {
            synchronized (obj1) {}
        }
    };
    static Thread t2 = new Thread() {
        public void run() {
            synchronized (obj1) {}
        }
    };
    public static void main(String[] args) {
        t1.start();
        t2.start();
    }
}

public class TraceProgram {
    static Object obj1 = new Object();
    static Object obj2 = new Object();
    static Thread t1 = new Thread() {
        public void run() {
            synchronized (obj2) {
                synchronized (obj1) {}
            }
        }
    };
    static Thread t2 = new Thread() {
        public void run() {
            synchronized (obj1) {
                synchronized (obj2) {}
            }
        }
    };
    public static void main(String[] args) {
        t1.start();
        t2.start();
    }
}

```

(a) Original trace program generated by CheckMate when LockPeeker is not integrated. (b) New trace program generated by CheckMate when LockPeeker is integrated.

Figure 7: The trace programs of SimpleBirt287102, generated by CheckMate with and without LockPeeker.

- Supplementing detected latent locks on objective code. We make latent locks visible to CheckMate by explicitly supplementing them as program replacements in the objective code along with the actual API call. In particular, when an API method `m` has no callback, we add program replacement before the call site of `m`. If `m` has callback, we firstly check if the callback is within the scope any latent lock, and surround it with corresponding lock (and conditions). Thus duplicated locks are added, but they will not affect the actual API execution, because when the added lock is reached, the thread must have hold the same lock already.

CheckMate executes the resulting program, and generates a new trace program (Figure 7b) which has the latent lock. As highlighted by the gray background, a new lock on `obj2` and relevant locking statements are newly generated. JPF thus can report the deadlock in BIRT-287102 by checking the new trace program.

Since the combination of LockPeeker and CheckMate advocates an idea of modifying the objective code, rather than modifying CheckMate itself, the resulting program with supplemented method replacements can also be used by other tools for detecting deadlocks. However, it still requires human efforts in matching locking scopes and callbacks (*i.e.*, determining into which locations the method replacements should be instrumented), we plan to automate the matching process in future.

4.3 RQ3. Significance of Threshold

As Section 3.2 explains, the blind exploration phase requires a threshold to terminate its exploration. Here we use different threshold values to evaluate the impacts of the threshold on different types of conditions.

4.3.1 Setup

Table 4: Significance of the threshold.

Project	Method	Condition Type	P_i	20		50		100		200		2500	
				C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
Tomcat	getServlet	1 nominal	51.2%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Derby	skip	1 numeric	9.9%	95%	0%	100%	100%	100%	100%	100%	100%	100%	100%
Derby	setSharedState	2 nominals	16.5%	100%	90%	100%	100%	100%	100%	100%	100%	100%	100%
Hsqldb	delete	2 numerics	1.5%	20%	0%	100%	0%	100%	0%	100%	0%	100%	40%
Derby	isValid	1 nominal + 1 numeric	5.4%	80%	2%	100%	41%	100%	51%	100%	47%	100%	45%
Tomcat	run	3 numerics	7.2%	1%	0%	23%	0%	87%	0%	98%	0%	98%	0%
Total			91.7%	81.8%	66.2%	86.2%	79.8%	90.8%	80.4%	91.6%	80.1%	91.6%	80.6%

Subjects. The impacts of the threshold highly depend on locking condition’s structures.

According to our analysis on the locks in Table 1, conditions on locks can be atomic conditions or compound ones. An atom condition can be a nominal condition that determines whether values equalize to constants, or a numeric condition in which numeric values are compared. A compound condition is composed of two or more atomic conditions. It has two types of information: (1) a condition structure for organizing two or more atomic conditions using logical operators, and (2) constant values indicating options of nominal conditions and boundary values of numeric ones.

We found that 334 out of 1,414 locks have conditions. Most conditions are simple, since 306 out of 334 have two or less atomic conditions. As compound locking conditions consisting of four or more atomic conditions are rare, we analyzed those consisting of at most three atomic conditions.

From the ten projects in Table 1, we investigated the significance of the threshold by selecting six methods. Each method has locking conditions. Thus we collected two atomic conditions, three compound conditions with two atoms, and a compound condition with three atoms. We simplified each method such that the method only keeps the locking conditions and locking actions. The representativeness of the samples can be calculated by

$$representativeness = \frac{condition_type}{total} \quad (5)$$

where *condition_type* is the number of locks in a type, and *total* the number of total locks with conditions (334 in our evaluation).

Metrics. We randomly searched the test instances. Following the guideline of Arcuri and Briand [2], for each threshold value, we executed our approach for one hundred times. We define two criteria to measure our results:

$$C_1 = \frac{structure_detected}{total_run} \quad (6)$$

$$C_2 = \frac{detected}{total_run} \quad (7)$$

where *structure_detected* is the times of correctly detecting the condition structures of locks, *detected* the times of correctly detecting both the condition structures and their boundary values, and *total_run* is one hundred.

4.3.2 Results.

Table 4 shows the results. Columns “Project”, “Method”, and “Condition Type” list the sampled projects, methods, and condition types. Column “ P_i ” lists the representativeness values that are calculated by Equation 5. Columns “20” to “2500” list C_1 and C_2 values calculated by Equations 6 and 7, when threshold values are set from 20 to 2500.

The results indicate that in blind exploration phase 100 is sufficient for detecting both structures and values for most condition

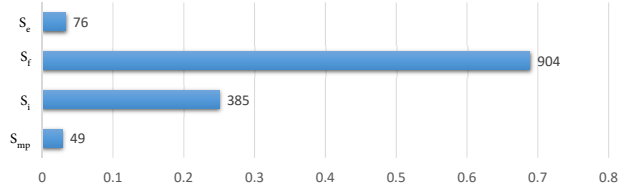


Figure 8: The distribution of locks of the ten open source projects. S_{mp} , S_i , S_f , and S_e denote the locks on method parameters, receivers, fields, and environment variables, respectively.

types (*i.e.*, the top five types). However, blind exploration alone cannot handle more complicated types, as it fails in detecting conditions for three numeric conditions.

To reveal the overall impact of a threshold, for C_1 and C_2 , their total values are calculated as:

$$total_C_1 = \sum P_i \times C_1 \quad (8)$$

$$total_C_2 = \sum P_i \times C_2 \quad (9)$$

When the threshold value is 50, the total values of C_1 and C_2 are 81.8% and 66.2%, respectively. When we chose some larger threshold values, the total values of C_1 and C_2 did not increase significantly. Thereafter, we selected 100 as the default threshold value, when the total values of C_1 and C_2 are 90.8% and 80.4%. The total value of C_2 only increases to 80.6% when the threshold is 2,500.

4.4 RQ4. Essential Variables

We next investigate which variables can trigger locks in API methods.

4.4.1 Setup

To answer which variables in test instances can cause locks inside API methods, we asked the three graduate students in Section 4.1 to classify locks into categories according to their origins. They analyzed all the locks in Table 1.

4.4.2 Result

Figure 8 shows the results. The 1,414 locks fall into the four categories:

- m ’s parameters, denoted as S_{mp} .
- An instance of class C (usually referred by `this`), or the class object of C (*i.e.*, $C.class$), denoted as S_i .
- Class C ’s fields, denoted as S_f .
- Instances, returned values from other classes, considered as environment variables, denoted as S_e .

Our study shows that the three types of locks such as S_f , S_i , and S_{mp} cover most locks: nearly two thirds (904 out 1,414) of

the locks are acquired on fields (S_f); more than a quarter (385 out of 1,414) of locks are acquired on their receivers (S_i); the remaining locks are acquired on environment variables (S_e) and method parameters (S_{mp}).

4.5 Threats to Validity

The threat to external validity includes the representativeness of our selected subjects. Our evaluations were conducted on the ten open-source projects. Although the ten projects have nearly one million lines of code and are widely used, our analyzed code is limited, and the data from other projects can be different. The threat could be further reduced by introducing more projects as our subject in future work, which shall cover more locking idioms.

The internal threat comes from the false negatives. In our evaluation, some locking candidates are omitted. Although rare, locks on S_e can still exist, leading to some false negatives. We plan to find a solution to exploring the environment variables for API methods and revealing latent locks on S_e in our future work.

5. RELATED WORK

Deadlock detection in APIs Researchers have tackled the problem of detecting deadlocks caused by third party libraries. Williams *et al.* [33] statically build lock-order graphs for both API code and client code, and then detects cycles in such graphs. Julia *et al.* [14] instrument both API code and client code to collect locking behaviors. From collected behaviors, they mine deadlock patterns, and avoid code from entering same patterns again. Samak and Ramanathan [26] take a multi-threaded library as input, and synthesize relevant multi-threaded tests from the input and analyze the associated execution traces for deadlocks. All the preceding approaches require accessible and analyzable API or native code, and thus are not able to detect deadlocks that involve API code. In the contrast, our approach detects latent locks in API code, allowing detecting corresponding deadlocks.

Deadlock detection For deadlock detection, execution models can be built via static or dynamic analysis. Static approaches [6, 23] employ various static analysis techniques (*e.g.*, call graph analysis and points-to analysis) to look for cyclic acquisitions in program source. Dynamic approaches [3, 4, 13, 21, 28] execute and record traces of a target program, and then analyze traces that are generated from non-deadlock execution to find potential deadlocks. The two types of approaches typically omit API or native methods, assuming these methods are irrelevant to deadlocks. Our approach complements the preceding approaches, since it detects latent locks in API or native methods.

API and multilingual code analysis The research on multilingual code analysis is related, since API code and source code are typically two types of code. Hong *et al.* [11] proposed new mutation operators in native code to locate faults for multilingual bugs. Kondoh and Onodera [16] statically analyze JNI programs to detect four types of bugs such as error checking, memory leak, invalid uses of a local reference, and JNI calls on critical regions. Li and Tan [19] work on exception checking for JNI. Lee *et al.* [17] focus on the debugging environment for multilingual programs. Tan [30] studies the define-use constraints for multilingual programs. Tang *et al.* [31] summarize API code to speed up code analysis. Henzinger *et al.* [9] infer automata from API code. Zhong *et al.* [34] infer specifications from API code, while Pandita *et al.* [24] infer specifications from both API documents and API code.

There is very limited research considering concurrency in multilingual code analysis. Li *et al.* [18] statically extract memory-

access models for both client code and native code, and inserts locks to guarantee atomicity.

Comparatively, our approach synthesizes locking code for API methods, complementing the preceding approaches.

6. DISCUSSION AND FUTURE WORK

As the first work on detecting latent locks in APIs, there are still adequate places for improvements. To provide insights for follow-up researchers, we carefully analyze the failures in Table 2, and we identify the following directions that need further exploration:

Call sequences and inputs When a target method requires sophisticated call sequences and input values to activate a lock acquisition, LockPeeker may not provide enough information to support such method invocations. Therefore, the target method execution may be ended too early to reach the locks, causing no false negatives. However, it is possible to support such situation by analyzing APIs' documents or searching code base for existing client code that calls such APIs to find necessary preconditions to assure method execution.

Repeated locks A thread can lock a resource multiple times in a method, sequentially or nestedly, but Algorithm 1 can detect only the first locking. Suppose a lock, L , is acquired twice in $thread_1$, and to reveal the second locking activity, it requires $thread_2$ to hold L just between the two acquisitions. In addition, for a nested situation, the acquisition in $thread_2$ will be blocked, if the object is already locked. Our current implementation still cannot handle the situations, but it may be feasible with the support of JDI¹⁰ (Java Debugging Interface).

Complicated condition branch LockPeeker cannot recognize complicated condition branches and conditions on complicated objects. For example, LockPeeker can discover locks inside `catch` clauses when the target method step into it, but cannot infer the corresponding try-catch clauses. As another example, it is common in Java to check conditions relevant to the returned value of a method invocation, but LockPeeker fails in recognizing such conditions.

7. CONCLUSION

Locks can be latent in API methods, which are not rare, but difficult to be detected. As existing approaches typically treat API methods as empty boxes, they are insufficient to detect deadlocks that involve latent locks in API methods. In this paper, we propose a novel approach, called LockPeeker, that detects latent locks in Java API methods by extensively testing a method, observing the locking behaviors, and inferring the locking structure and conditions. Our evaluation results have demonstrated that LockPeeker is capable of derive a clear locking tree for an API method with a small set of test instances. We believe that developers can use LockPeeker to identify the latent locks in API methods and improve the robustness of Java applications.

8. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments. We would also thank Weizhao Yuan, Yingyi Wang and Xuliang Liu for their help in analyzing subject projects. This work is sponsored by the 973 Program in China (No. 2015CB-352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102), and the grant of Science and Technology Commission of Shanghai Municipality (No. 15DZ1100305).

¹⁰<http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html>

9. REFERENCES

- [1] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji. Mutation of java objects. In *ISSRE*, 2002.
- [2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, 2011.
- [3] Y. Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *ICSE*, 2012.
- [4] Y. Cai, S. Wu, and W. K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE*, 2014.
- [5] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Integrated dynamic and static analysis for atomicity violations. In *Proc. FASE*, 2009.
- [6] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.
- [7] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.
- [8] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, 2000.
- [9] T. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE*, 2005.
- [10] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.
- [11] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. Mutation-based fault localization for real-world multilingual programs (T). In *ASE*, 2015.
- [12] D. Hovemeyer and W. Pugh. Finding concurrency bugs in Java. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.
- [13] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.
- [14] H. Julia, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.
- [15] A. Kalinovsky. *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. Pearson Higher Education, 2004.
- [16] G. Kondoh and T. Onodera. Finding bugs in java native interface programs. In *ISSTA, ISSTA '08*, 2008.
- [17] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley. Debug all your code: portable mixed-environment debugging. In *OOPSLA*, 2009.
- [18] S. Li, Y. D. Liu, and G. Tan. JATO: native code atomicity for Java. In *APLAS*, 2012.
- [19] S. Li and G. Tan. JET: exception checking in the java native interface. In *OOPSLA*, 2011.
- [20] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *ASE*, 2015.
- [21] Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *ICST*, 2011.
- [22] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. GRT: an automated test generator using orchestrated program analysis. In *ASE*, 2015.
- [23] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.
- [24] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *ICSE*, 2012.
- [25] S. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):660–674, 1991.
- [26] M. Samak and M. K. Ramanathan. Omen+: A precise dynamic deadlock detector for multithreaded Java libraries. In *FSE*, 2014.
- [27] N. Shafiei and F. v. Breugel. Automatic handling of native methods in Java PathFinder. In *Proc. SPIN*, 2014.
- [28] F. Sorrentino. Picklock: A deadlock prediction approach under nested locking. In *SPIN*, 2015.
- [29] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proc. FSE*, 2010.
- [30] G. Tan. JNI light: An operational model for the core JNI. In *APLAS*, 2010.
- [31] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, 2015.
- [32] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
- [33] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, 2005.
- [34] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented APIs from API source code. In *Proc. APSEC*, 2008.